Testing, Vectors, and Grids

MONDAY, APRIL 7

In this lecture, we will introduce use of **`Vector`** and **`Grid`** and talk about what testing fundamentals look like in CS106B.

- 📚 Readings: Text 2.5-2.9, 5.1 and class documentation for Vector and Grid
- 📝 Lecture quiz on Canvas
- 🎬 Lecture video on Canvas

Contents

1. Announcements

2. Warmup and Reinforcement

3. Design Principles: Functional Decomposition and More

4. Testing with `SimpleTest.h`

5. Abstract Data Types (ADTs): Introduction and Overview

6. `Vector` Overview

7. `Vector` Functions and Sample Code

8. Runtime Comparison: `add(value)` vs. `insert(0, value)`

9. `Grid` Overview

10. `Grid` Functions and Sample Code

11. Four Ways to Print Grid Contents

12. Common Pitfalls

13. What's next?

14. Exam Prep

Announcements

- Please be sure to glance through the following documents sometime this week:
  - Recommended Settings for Qt Creator ← Lots of quality-of-life improvements!
  - Assignment Submission Checklist
  - Style Guide
- Section assignments will be available tomorrow (Tuesday) by 5 PM or so. If you missed section sign-up or want to swap swap sections, that will be possible in Paperless! Section starts **this week**, and section attendance counts for 5% of your grade.
- Be sure you're reading Jonathan's weekly post on Ed.
- Your first quiz series is due this Wednesday at 1 PM.
- Have a burning question during lecture? Jonathan is creating a megathread at the start of each lecture where you can get responses from him in real time. This is great for questions that might not be relevant to the whole class or questions that we just don't have time to get to because of the pace at which we're covering material.
- Please head to LaIR for assignment help early. LaIR gets packed the day of and before each assignment deadline. Avoid frustration and long wait times by starting early and getting help early. 🙂

Warmup and Reinforcement

We started with a program designed to determine whether a given username appears in a given password string -- and to print an alert if so:

> **Warning: This code suffers from several major design issues. See the following section of notes ("Design Principles") for more.**

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
   string s = "Sean";
   string p = "sean'spassword";

   // How can we determine whether the given username (s) appears in the password (p)?

   if ( ??? )
   {
      cout << "Uh oh! Bad password." << endl;
   }
   else
   {
      cout << "Hooray!" << endl;
   }

   return 0;
}
```

The solution, which we found by glancing through the functions listed in the `strlib.h` docs, is:

> **Warning: This code suffers from several major design issues. See the following section of notes ("Design Principles") for more.**

```
#include <iostream>
#include "console.h"
#include "strlib.h"
using namespace std;

int main()
{
   string s = "Sean";
   string p = "sean'spassword";

   // How can we determine whether the given username (s) appears in the password (p)?

   if (stringContains(toLowerCase(p), toLowerCase(s)))
   {
      cout << "Uh oh! Bad password." << endl;
   }
   else
   {
      cout << "Hooray!" << endl;
   }

   return 0;
}
```

We then augmented this to check whether the username appears in the password string if we extract just the alphabetic portion:

> **Warning: This code suffers from several major design issues. See the following section of notes ("Design Principles") for more.**

```
#include <iostream>
#include "console.h"
#include "strlib.h"
using namespace std;

int main()
{
   string s = "Sean";
   string p = "s1e1a1n1'1s1p1a1s1s1w1o1r1d";
   string p0;

   for (int i = 0; i < p.length(); i++)
   {
      if (isalpha(p[i]))
      {
         p0 += p[i];
      }
   }

   if (stringContains(toLowerCase(p0), toLowerCase(s)))
   {
      cout << "Uh oh! Bad password." << endl;
   }
   else
   {
      cout << "Hooray!" << endl;
   }

   return 0;
}
```

Design Principles: Functional Decomposition and More

The code above suffers from several flaws that make it difficult to read and, therefore, difficult to debug, augment, and maintain. Issues include:

- We are not using meaningful variable names.
- Everything is dumped into `main()`, leading to code that is difficult for someone who's not familiar with the code to parse through and debug. As written, if our program isn't working correctly, it can be hard to determine where things are going wrong, because there's so much happening all in one place. It's also hard to subject individual components of the code to robust testing when they're all slopped into `main()` like this.
- There are no comments to assist the reader in figuring out what's going on. However, the ideal scenario would be to rewrite the code in a way that makes comments somewhat unnecessary.

We solved the broader structural issues by moving chunks of code into functions that serve a single purpose and naming those functions with verb phrases that conveyed what they do.

(*Key Concept!*) Breaking up a task into constituent functions that each fulfill a single purpose is called **functional decomposition** and improves the readability and maintainability of our code.

```
#include <iostream>
#include "console.h"
#include "strlib.h"
using namespace std;

string extractAlpha(string s)
{
   string result;

   for (int i = 0; i < s.length(); i++)
   {
      if (isalpha(s[i]))
      {
         result += s[i];
      }
   }
   return result;
}

bool passwordChecksOut(string username, string password)
{
   string alphaPortion = extractAlpha(password);
   return !stringContains(toLowerCase(alphaPortion), toLowerCase(username));
}

int main()
{
   string username = "Sean";
   string password = "s1e1a1n1'1s1p1a1s1s1w1o1r1d";

   if (passwordChecksOut(username, password))
   {
      cout << "Hooray!" << endl;
   }
   else
   {
      cout << "Oh no! Bad password." << endl;
   }

   return 0;
}
```

This code is now much more readable, and we have a few units that can be independently and rigorously tested: the `extractAlpha()` and `passwordChecksOut()` functions. The introduction of meaningful variable names and function names also means that we needn't bog down our code with comments explaining what's going on.

A nice side effect of this approach is that we now have one place where all the password-checking logic is neatly bundled together and isolated from the rest of the code. If we want to add new restrictions (such as requiring the password be at least 8 characters long), there is a very clear place to add the code for that: the `passwordChecksOut()` function.

Testing with `SimpleTest.h`

At this point, we turned our attention to testing. While introducing this topic today, I mentioned test-driven development.

The Stanford C++ Library has a powerful testing framework that you can use with just a few simple steps:

1. `#include "SimpleTest.h"` in your code
2. Write your own tests using `STUDENT_TEST("description of test") { ... }`
3. Populate each `SIMPLE_TEST` with `EXPECT_EQUAL( ... )` statements that verify the results of various operations.
4. Call `runSimpleTests(ALL_TESTS)` to cause the tests to be run.

Further information about all these steps -- as well as information about other testing functionality that is available to you -- can be found in the SimpleTest Guide.

Note that in programming assignments, you can (and should!) make use of `STUDENT_TEST` to add your own test cases, but you should be careful not to modify any test we have provided for you. Those are created using `PROVIDED_TEST` instead of `STUDENT_TEST`.

(*Key take-away!*) In discussing things to test for in `extractAlpha()`, we came up with all kinds of scenarios:

- What if all the characters are alphabetic?
- What if there are non-alpha characters are at the end of the string?

- What if there are non-alpha characters are at the beginning of the string?
- What if there are non-alphabetic characters in the middle of the string?
- What if there are alternating alphabetic and non-alphabetic characters?
- What if we have only non-alphabetic characters, and no alphabetic ones?
- What if our non-alphabetic characters are not digits, but other symbols ('$', '*', etc.)?
- What if our string consists entirely of whitespace characters ('\n', spaces, and so on)?
- What if we have an empty string (a string with zero characters)?
- What if we have a string with exactly 1, 2, or 3 characters?
- What if we have a very long string?
- What if our string is even in length? What if the length is odd?

These are the sorts of things we hope you will be thinking about as you develop test cases for your own programs this quarter.

Here is the resulting code with test cases for the `extractAlpha()` function.

```
#include <iostream>
#include "console.h"
#include "SimpleTest.h"
#include "strlib.h"
using namespace std;

string extractAlpha(string s)
{
   string result;

   for (int i = 0; i < s.length(); i++)
   {
      if (isalpha(s[i]))
      {
         result += s[i];
      }
   }
   return result;
}

bool passwordChecksOut(string username, string password)
{
   string alphaPortion = extractAlpha(password);
   return !stringContains(toLowerCase(alphaPortion), toLowerCase(username));
}

STUDENT_TEST("various tests of extractAlpha() function")
{
   EXPECT_EQUAL(extractAlpha("sean"), "sean");
   EXPECT_EQUAL(extractAlpha("sean11"), "sean");
   EXPECT_EQUAL(extractAlpha("1sean"), "sean");
   EXPECT_EQUAL(extractAlpha("s1ean"), "sean");
   EXPECT_EQUAL(extractAlpha("s1e1a1n"), "sean");
   EXPECT_EQUAL(extractAlpha("s9e$a***&n"), "sean");
   EXPECT_EQUAL(extractAlpha(""), "");
   EXPECT_EQUAL(extractAlpha("9$&"), "");
}

int main()
{
   runSimpleTests(ALL_TESTS);

   string username = "Sean";
   string password = "s1e1a1n1'1s1p1a1s1s1w1o1r1d";

   if (passwordChecksOut(username, password))
   {
      cout << "Hooray!" << endl;
   }
   else
   {
      cout << "Oh no! Bad password." << endl;
   }

   return 0;
}
```

Abstract Data Types (ADTs): Introduction and Overview

Next, we waded into a unit on ADTs (abstract data structures).

The implementation details of an ADT might vary greatly from one language to another, and there are often many different ways to implement a particular ADT in any one language, as well. Having a set of data structures that

provide an agreed-upon model for how data is stored and manipulated means we can:

> (a) have a common language for discussing solutions to a wide range of problems with other people in our field
>
> (b) write code that is fairly comprehensible even when viewed by someone who isn't familiar with the specifics of the language it's written in
>
> (c) focus on problem-solving without getting bogged down in unnecessary, low(ish)-level implementation details

As we explore ADTs initially, we will approach them from the client perspective. That is, we will act as consumers of ADTs that have been implemented for us in C++ with a focus on *what* those ADTs allow us to accomplish, without getting bogged down with the behind-the-scenes details of exactly *how* those ADTs are implemented. This will rapidly open up entire vistas of new problems that we can solve with code.

We will, however, return to the implementation details of these data structures later this quarter.

`Vector` Overview

The first ADT we explored was the Vector:

- A vector is a type of list that expands and shrinks automatically as new elements are added or removed.
- The elements in a vector are inherently ordered. Each element in a vector is assigned a position relative to the others.
- The elements in a vector are indexed 0 through *n* - 1 (where *n* is the number of elements in the vector).
- The vector is a homogenous container. All the elements in a given vector must be of the same type.

Underlying the Vector from the Stanford C++ Libraries is an array, which means:

- Elements are stored in contiguous blocks in memory.
- We can quickly access the element at any given index in the the vector.

The vector we are describing is, of course, analogous to a list in Python or an ArrayList in Java. This ADT is ubiquitous in the real world. One place where you see a vector in action pretty much daily is if you open multiple tabs in your web browser. Tabs have an inherent ordering, and the size of the underlying data structure used to keep track of them can grow and shrink as you open and close more tabs.

To create a vector, we must include the following library:

```
#include "vector.h"
```

The syntax for creating a vector is as follows:

```
Vector< DATA_TYPE >  VARIABLE_NAME ;
```

Optionally, we can provide an initial list of elements that the vector contains:

```
Vector< DATA_TYPE >  VARIABLE_NAME  { ELEMENT_1 ,  ELEMENT_2 , ...,  ELEMENT_N };
```

(*Important note!*) C++ has a built-in vector that we will not be using this quarter. The C++ vector has a lowercase 'v'. The vector from the Stanford C++ Libraries has an uppercase 'V'.

`Vector` Functions and Sample Code

`myVector[i]` accesses the element at index `i` .

Additionally, we have the following functions:

Member Function	Description
size()	returns number of elements in the vector
isEmpty()	returns `true` if the vector is empty, `false` otherwise
add(value)	adds a new element to the end of the vector

Member Function	Description
insert(index, value)	inserts the value before the specified index, moving the values after it up by one index
remove(index)	removes the element at the specified index, moving the rest of the elements down by one index
clear()	removes all elements from the vector
. . .	For an exhaustive list, see: Stanford **`Vector`** class

Here's a slightly fleshed-out version of the example we saw in class today:

```
#include <iostream>
#include "console.h"
#include "vector.h"
using namespace std;

int main()
{
   Vector<int> v = {15, 20, 18};

   // A this point, we have:
   // +----+----+----+
   // | 15 | 20 | 18 |
   // +----+----+----+
   //   0    1    2

   // We have a size() function that tells us how many elements are in a vector,
   // and we can access individual elements using VARIABLE_NAME[INDEX]. Putting those
   // together, we can loop through the vector and print its contents:

   for (int i = 0; i < v.size(); i++)
   {
      cout << "index " << i << ": " << v[i] << endl;
   }

   // We can also check whether a vector is empty using the isEmpty() function:

   if (v.isEmpty())
   {
      cout << "Empty vector." << endl;
   }

   // The add() function adds an element to the end of the vector.

   v.add(33);

   // We now have:
   // +----+----+----+----+
   // | 15 | 20 | 18 | 33 |
   // +----+----+----+----+
   //   0    1    2    3

   // The insert() function adds an element at a given index, first scooting over
   // all elements at and to the right of that index in order to make room for the
   // new element.

   v.insert(2, 90);

   // We now have:
   // +----+----+----+----+----+
   // | 15 | 20 | 90 | 18 | 33 |
   // +----+----+----+----+----+
   //   0    1    2    3    4

   // The remove() function removes the element at a given index, then scoots over
   // all elements to the right of that index so as not to leave a gap in the vector.

   v.remove(0);

   // We now have:
   // +----+----+----+----+
   // | 20 | 90 | 18 | 33 |
   // +----+----+----+----+
   //   0    1    2    3

   return 0;
}
```

Runtime Comparison: `add(value)` **vs.** `insert(0, value)`

We saw that inserting at the beginning of a vector with `v.insert(0, value)` forces us to spend time scooting over every single element that the vector already contains. Repeatedly inserting at the beginning of a vector, then, is dangerously slow. Each new insertion must perform more work than the last, because there is one more element to scoot over each time.

The `v.add(value)` function is, in comparison, quite fast. It just plunks a new element onto the end of the vector. This occasionally causes the vector to quietly expand behind the scenes (a detail we will explore later this quarter), but is otherwise a fast operation.

To demonstrate the enormous runtime disparity between repeated calls to `v.insert(0, value)` and `v.add(value)`, I produced the following code and used `TIME_OPERATION` from our SimpleTest library to see how long `vectorAdd()` and `vectorInsert()` were taking on various input sizes:

```
#include <iostream>
#include "console.h"
#include "SimpleTest.h"
#include "vector.h"
using namespace std;

void vectorAdd(int size)
{
   Vector<int> v;

   for (int i = 0; i < size; i++)
   {
      v.add(i);
   }
}

void vectorInsert(int size)
{
   Vector<int> v;

   for (int i = 0; i < size; i++)
   {
      v.insert(0, i);
   }
}

PROVIDED_TEST("runtime comparison")
{
   int size = 500000;

   TIME_OPERATION(size, vectorAdd(size));
   TIME_OPERATION(size, vectorInsert(size));
}

int main()
{
   runSimpleTests(ALL_TESTS);
   return 0;
}
```

We saw that the time it takes to execute `vectorInsert()` is **EXPLOSIVE**, and the difference between that and the time it takes to execute `addInsert()` is absolutely astounding -- especially as `size` increases.

Our in-class experiments yielded the the following runtimes:

size	vectorAdd() runtime	vectorInsert() runtime	how much slower?
50,000	0.005s	0.086s	17.2x
500,000	0.029s	9.794s	337.8x 😳

`Grid` Overview

We then saw the Grid ADT, which is effectively a rectangular array. You could conceive of it as a vector of vectors. Grids can be used to represent all kinds of things, including:

- spreadsheets (where element in a grid corresponds to a cell in a spreadsheet)
- game boards (e.g., an 8x8 grid of characters indicating locations of pieces on a chess board)
- images (a grid of color codes, one for each pixel in an image)
- ... and so many more

Like a vector, the elements in a grid are ordered and indexed. A grid has some number of rows (horizontal) and

columns (vertical).

To create a grid, we must include the following library:

```
#include "grid.h"
```

The syntax for creating a vector is as follows:

```
Grid< DATA_TYPE >  VARIABLE_NAME ( NUM_ROWS ,  NUM_COLUMNS );
```

(*Important note!*) We must capitalize the 'G' in "Grid," just as we capitalize the 'V' in "Vector."

(*Key take-away!*) We saw that this data structure is "row-major" because the elements within a row are clustered together in memory. If you can remember the phrase "row-major," you can remember that the row, being such a "major" aspect of the grid's structure, always comes before the column -- both when creating a grid and accessing individual elements within a grid.

`Grid` Functions and Sample Code

`myGrid[i][j]` accesses the element at row `i`, column `j`.

Additionally, we have the following functions:

Member Function	Description
numRows()	returns the number of rows in the grid
numCols()	returns the number of columns in the grid
resize(rows, cols)	changes the dimensions of the grid and re-initializes all entries to their default values
inBounds(row, col)	returns `true` if the specified (row, col) position is in the grid, `false` otherwise
. . .	For an exhaustive list, see: Stanford `Grid` class

Here's a fleshed-out version of the example we saw in class today:

```
#include <iostream>
#include "console.h"
#include "grid.h"
using namespace std;

int main()
{
   Grid<int> g(3, 4);

   // Values in an integer grid are zeroed out by default. At this point, we have:
   //     +----+----+----+----+
   //  0  | 0  | 0  | 0  | 0  |
   //     +----+----+----+----+
   //  1  | 0  | 0  | 0  | 0  |
   //     +----+----+----+----+
   //  2  | 0  | 0  | 0  | 0  |
   //     +----+----+----+----+
   //       0    1    2    3

   // We can update the value at row 2, column 3, like so (see corresponding
   // cell highlighted in the diagram below):
   g[2][3] = 18;

   // We now have:
   //    +----+----+----+----+
   //  0 | 0  | 0  | 0  | 0  |
   //    +----+----+----+----+
   //  1 | 0  | 0  | 0  | 0  |
   //    +----+----+----+----+
   //  2 | 0  | 0  | 0  | 18 |
   //    +----+----+----+----+
   //      0    1    2    3

   // Print an entire grid. For each row, print the values in each column.

   for (int i = 0; i < g.numRows(); i++)
   {
      for (int j = 0; j < g.numCols(); j++)
      {
         cout << g[i][j];

         // If not the last element in this row, print a comma and a space.
         if (j != g.numCols() - 1)
         {
            cout << ", ";
         }
      }
      cout << endl;
   }

   return 0;
}
```

Four Ways to Print Grid Contents

Here are four ways to print the contents of a grid to the screen:

1. We can use nested for loops to display the contents of a grid in the same order as above.
2. (*Not mentioned in class; very good to be aware of.*) We can use a for-each loop (i.e., "enhanced for-loop" or "range-based loop") to loop through and display the contents of a grid. These are processed in a row-major fashion. Note that this approach does not give us access to the row and column of the element being processed within the loop.
3. (*Not mentioned in class; very good to be aware of.*) We can loop through all the valid coordinates in a grid as `GridLocation` objects, and those can be used to access specific grid elements.
4. (*Not mentioned in class; very good to be aware of.*) We can send a grid directly to `cout` and get a nicely formatted string that conveys the grid contents. This approach prints contents in a row-major fashion, as well.

```
#include <iostream>
#include "console.h"
#include "grid.h"
using namespace std;

int main()
{
   Grid<int> g(2, 3);

   // Just random values. :)
   g[0][0] = 41;
   g[0][1] = 53;
   g[0][2] = 98;
   g[1][0] = 18;
   g[1][1] = 21;
   g[1][2] = 16;

   // Option #1: For-Each Loop
   cout << "Grid for-each loop:" << endl;
   for (int i : g)
   {
      cout << i << endl;
   }
   cout << endl;

   // Option #2: Nested for loops.
   cout << "Grid nested loops:" << endl;
   for (int row = 0; row < g.numRows(); row++)
   {
      for (int col = 0; col < g.numCols(); col++)
      {
         cout << g[row][col] << endl;
      }
   }
   cout << endl;

   // Option #3: Iterate over grid locations.
   cout << "Grid locations:" << endl;
   for (GridLocation loc : g.locations())
   {
      cout << loc << " -> " << g[loc] << endl;
   }
   cout << endl;

   // Option #4: Just dump directly to cout.
   cout << "Grid contents:" << endl;
   cout << g << endl << endl;

   // Example of manual specification of GridLocation.
   GridLocation myLoc(0, 1);
   cout << g[myLoc] << endl;

   return 0;
}
```

output:

```
Grid for-each loop:
41
53
98
18
21
16

Grid nested loops:
41
53
98
18
21
16

Grid locations:
r0c0 -> 41
r0c1 -> 53
r0c2 -> 98
r1c0 -> 18
r1c1 -> 21
r1c2 -> 16

Grid contents:
{{41, 53, 98}, {18, 21, 16}}

53
```

The syntax for passing a vector of integers by reference is as follows:

```
void printVector(Vector<int>& v)
{
   ...
}
```

Common Pitfalls

(*Not mentioned in class; very good to be aware of.*) A common pitfall with both vectors and grids is to access indices that are out of bounds. This will cause your program to crash spectacularly:

```
#include <iostream>
#include "console.h"
#include "vector.h"
using namespace std;

int main()
{
   Vector<int> v = {10, 20, 30};
   v[3] = 40;

   return 0;
}
```

output:

```
*** STANFORD C++ LIBRARY
*** The Day04Lecture program has terminated unexpectedly (crashed)
*** A fatal error was reported:

    Vector::operator []: index of 3 is outside of valid range [0..2]

*** To get more information about a program crash,
*** run your program again under the debugger.
```

What's next?

On Wednesday, we continue our discussion of ADTs. We will delve into the `Stack` and `Queue` containers from the Stanford C++ Libraries.

If you're the sort of person who likes to read ahead, check out the course schedule in the Living Omni-Grid, which now lists what chapters/sections of the textbook we will cover in each lecture.

Exam Prep

1. We wrote various tests for the `extractAlpha()` function in class today. Add to that code a separate `STUDENT_TEST` that contains a suite of test cases for the `passwordChecksOut()` function. What sorts of things would you test to ensure you have robust testing coverage for that function?

2. Write a function that takes a vector of integers as its only argument and prints it in the following format:

```
+----+----+----+----+----+----+
| 43 | 19 | 12 | 95 | 83 | 13 |
+----+----+----+----+----+----+
  0    1    2    3    4    5
```

The function signature is:

```
printVector(Vector<int>& v)
```

To simplify the problem, you may assume all the integers in the vector have exactly two digits.

Additional challenge (this is fairly advanced; don't worry too much about this one if you don't have time):

- How could you ensure when printing the contents of a vector that the largest box has exactly enough space for the largest integer in the vector to be surrounded by one space on either side, and all other boxes have the same amount of space, with each integer being centered within its cell to the extend possible? For example:

```
+------+------+------+------+------+------+
|  43  |  1   |  9   | 129  | 5821 |  3   |
+------+------+------+------+------+------+
   0      1      2      3      4      5
```

3. In the previous problem, why would we pass the vector by reference if we have no intention of modifying it in the `printVector()` function? (Hint: See the notes from Friday, in the "Pass-by-Reference Functions" section.)

Highlight for solution to #3: We often pass containers to functions by reference simply to save time and space (i.e., memory). If we pass a HUGE vector to a function by value, our function has to create a whole separate copy of that vector's contents, which takes time in addition to using up extra memory (since we'll have a second copy of the vector hanging around in memory). Passing by reference is a smaller, faster transaction and is sometimes done simply for the sake of efficiency.

4. As always, see the following resources for additional practice problems and reinforcement of this material:

- The course reader (free PDF available online; see readings listed at the top of today's notes).
- Today's lecture quiz.
- Assignment 1.
- Problems covered in section.

Website programming by Julie Zelenski with modifications by Sean Szumlanski • Styles adapted from Chris Piech • This page last updated 2025-Apr-07